Finding shared RSA factors in the Certificate Transparency logs

Henry Faltin Våge

Master's Thesis
Department of Informatics,
University of Bergen

May 13, 2022

Acknowledgements

Thank you to my friends and family for all the helpful insights and discussions. I also want to thank Mariah for her support and help throughout this time.

Thanks to my fellow students at Simula UiB for all the help and for creating such a great environment for discussions.

Finally a special thanks to my excellent supervisor Håvard for all your help with this thesis. Through periodic meetings and discussions it has been an honour working with you.

Henry Faltin Våge
Bergen, 22 juni 2022

Abstract

When generating RSA keys, proper random generators are crucial. If the generators are not truly random, keys may be generated with the same factors, making them vulnerable to compromise. Doing a simple greatest common divisor computation would reveal the secret factors. We collected over 159 million unique RSA public keys from the Certificate Transparency logs, which is, to our knowledge, the largest set used for such an analysis so far. Our goal was to check if any of these keys shared factors, thus allowing us to compute the private keys easily. To do this, we implemented a batch greatest common divisor algorithm used for this purpose in previous studies.

Our result from checking the 159 million RSA keys was that we factored eight keys, all of which were issued by the same certificate authority. We then gathered more than 700,000 keys from that particular certificate authority, of which we were able to factor 355 keys. We reached out to the issuer of the broken certificates, and they launched an investigation into our findings. Their investigation concluded that all broken keys were generated by a single user who they claim had abused their system.

Contents

Chapter 1

Introduction

In this chapter we will cover some relevant history and terminology as well as the general area in which we will operate.

1.1 History

Cryptography has been used since the time of the Roman empire to conceal or hide messages from prying eyes. It is historically tied to governments and military services to communicate securely. In order to achieve this, two parties need to have a shared secret or key. The parties can then use this key to encrypt some message, called plaintext, by scrambling the message into something meaningless, called a ciphertext. The other party that receives this ciphertext can then decrypt the ciphertext with the key to recover the message.

One example of an old encryption algorithm is the Caesar cipher. The cipher was used in the Roman empire and today is considered basic. It works by shifting the alphabet by some k positions to the right. For instance, if $k = 2$ then the plaintext "abc" would encrypt to the ciphertext "cde" [33]. Since then, cryptography has evolved to ensure that it is as hard as possible for anyone not knowing the key to figure out the plaintext given the ciphertext.

By the time of the second world war in 1939, cryptography was widely used and deployed by military services. By then, cryptography was more advanced and relied on portable mechanical machines to perform encryption and decryption such as the famous Enigma machine. As the main principle had not changed, namely that the parties that communicated had to possess the same key, this required significant logistics. Therefore each soldier or spy on the field needed the same key as the one receiving their messages. The distribution and replacing of keys is called key distribution. Key distribution is one of the main drawbacks of symmetric cryptography. The main problem is scaling, because communication between two parties is quite easy, but what if hundreds or thousands of people need to communicate with each other? Then key distribution becomes a demanding task. As the key needs to be secret, it is costly to distribute the keys.

As computers became more common and connected throughout the ’80s and ’90s, the desire to communicate securely online had clear benefits. This development also meant that the scale of communicating parties exploded. Now key distribution was a

limiting factor.

Asymmetric cryptography is relatively new and was discovered in the '70s and is different from symmetric cryptography, which is the type of cryptography we have discussed so far. The main difference is that asymmetric cryptography has two keys, one for encrypting (public key) and one for decrypting (private key). Also, only knowing the public key, it should be practically impossible to find the private key. This is very useful as one could give the public key to everyone in the world, who could then send encrypted messages only you can decrypt.

Asymmetric cryptography could in theory replace symmetric cryptography but as symmetric cryptography is much faster therefore we still use it every day. Asymmetric cryptography is therefore used when smaller amounts of information should be encrypted, such as encrypting a symmetric key. This is done by encrypting the symmetric key with a public key and sending it to the owner of the private key. Now the owner can decrypt the key and start using symmetric encryption. The combination of the two methods provide cryptographers with great tools to build complex systems achieving secure communication between parties that have never communicated with each other before. We will look at some of them in this thesis.

As the world became more interconnected the trust online started to decrease, as people and systems were vulnerable to hacking. The field of cybersecurity is about protecting systems and information. Here we define some of the basic objectives when protecting information.

Confidentiality refers to keeping information secret, for instance passwords, medical journals, sensitive information etc. should be confidential. This is something most people can relate to, by not wanting to share private information with third parties. Confidentiality is the original purpose of encryption, to conceal secret information so third parties can not read them. This is commonly achieved by using either symmetric or asymmetric encryption.

Integrity refers to the certainty and accuracy of some information, such as a document with your signature on it. If it is possible to forge your signature or change the document after you signed it, the document would not have integrity. So integrity is the certainty that the content of a message has not changed in transit between its sender and receiver. Integrity can be achieved by using digital signatures, which is one form of asymmetric cryptography. Digital signature users have a private key, and as long as the key is secret, the only person who could have signed a document is the owner of the key.

Authentication is the process of verifying the identity of an entity. This is often an initial step in a process for obtaining sensitive information. For example by logging into your bank, you have to be authenticated to ensure you have privileges to obtain your banking information and make payments. As we will see in Chapter 2, certificates play an important role in authentication.

Non-repudiation means it should not be possible to send a message and later claim you did not send it [33]. This is an important feature when building complex systems, as this helps in achieving transparency and responsibility. Non-repudiation can also be achieved by using digital signatures.

We will now see how asymmetric and symmetric cryptography help us in achieving these goals.

1.1.1 Symmetric Cryptography

We will not go into details about symmetric cryptography, but cover what is relevant for this thesis. Symmetric cryptography is as previously mentioned cryptography where the same key is used for encryption and decryption. This is the fastest way to do encryption today, and one example of such a symmetric encryption algorithm is the Advanced Encryption Standard (AES) [10]. It is important that the encryption and decryption is fast, as the amounts of data in modern applications is ever increasing. As an example the quality of pictures and videos keep getting better and their file size keep increasing. The rise of cloud computing and storage also means large amounts of traffic. Symmetric encryption ensures confidentiality as long as the keys are managed securely. The drawback of symmetric cryptography is the need for key distribution.

Message Authentication Codes

A Message Authentication Code (MAC) is a way of authenticating the origin of a message using symmetric cryptography. This is done by two parties again sharing a common secret key. A MAC algorithm takes a secret key and the message as input, and produces an output called a tag. This way the tag depends on the secret key. So when a sender sends a message along with a tag, the recipient calculates the tag using the message and the same secret key and compares the computed tag with the received one. If they are equal the recipient is sure that the sender possesses the same secret key and that the received message is exactly the same as the one that was sent. That is, MAC ensures authentication and integrity of messages, based on a pre-shared key [33].

1.1.2 Asymmetric Cryptography

Asymmetric cryptography is as previously mentioned cryptography where two different keys are used. The public key is used for encryption and the private key is used for decryption. This is slightly misleading particularly for RSA as both keys actually can decrypt and encrypt, but the names simply refer to which key is secret and which key is public.

Asymmetric cryptography is used for many purposes, including confidentiality, authentication, and integrity. One example of an asymmetric encryption algorithm is RSA, which we will discuss in more detail in Chapter 3. Today asymmetric cryptography is mostly used for key exchange using certificates and signatures. We will discuss certificates further in Chapter 2.

Digital Signatures

Digital signatures are a way of signing some data digitally using asymmetric cryptography. Digital signatures has some resemblance to MACs. A digital signature is done by "encrypting" a message with a private key, then sending this "ciphertext" along with the original message. This way a recipient can decrypt the "ciphertext" using the public key and verify that the decrypted message is identical to the original message. To get the decrypted message to be identical to the original one, should only be possible, if the sender has the private key. If you are certain that only one particular person has the private key and signs with it, then using a digital signature authenticates this person.

Integrity comes from the fact that the computed and received signatures should be identical. Non-repudiation and authentication is achieved when the only person able to sign, is the person with the private key. Therefore a digital signature provides authentication, integrity and non-repudiation. This is why, today, we can sign documents online and have them count as legal signatures [33].

1.2 Cryptography today

From previously being used mainly by governments and military services around the world, cryptography today is used by almost all people on the internet [18]. It is still applied in governments and the military but is now also used in almost all civilian electronic communications, including mobile communication, web browsing, email traffic and more.

One reason, for the use of encryption, is that most information that we share and communicate is considered private or non-public. There are obvious things we would like to ensure are confidential and have integrity, such as online banking and private communications. However, some information may still be considered public such as a blog or a news site, but could still use encryption in transit to protect businesses and people's privacy. Another reason for the use of encryption today is that it is considered a good security practice as we will see in Chapter 4.

Additionally the information can considered public today, such as reading the news and browsing the web, may tell a lot about you. The advertisement business is a good example on how this data is being used to send us ads more accurately. The danger here is when a lot of public or private information is systematized and aggregated to the point where queries like "what are the interests of this specific individual?" can be answered with a high degree of accuracy. So knowing you read a specific article online may not tell that much about you, but when considering every article you have read the past year, this may form a pattern. This is not possible if all our digital activities are confidential, and more importantly in this setting, anonymous. Today this is possible because the services we use track, buy and sell this information in order to get paid for showing advertisements to us [15]. If encryption was not used, this information could be tracked and stored by Internet Service Providers or any other link between you and the site you are visiting.

Our private conversations are perhaps one of the most important communications where we want confidentiality and integrity. This could be for many reasons, but from a security perspective, this is where we share our most private information. This is also where we have the highest level of trust, so integrity is crucial, as messages from our friends and family has trust that could be misused by a third party. So most messaging services use encryption, such as email and messaging apps. The messaging platforms want to provide the best service, so using encryption is now seen as a criteria for some people in order to use their platform. We can see this as several of the most popular chatting apps use encryption in their services, and promote the encryption in their campaigns. There is also more laws implemented, mainly to protect privacy, by encrypting personal data.

Practically all cryptography today happens digitally, primarily by standardized protocols that have been thoroughly scrutinized and tested. The field of cryptography can

be considered both theoretical and practical. Cryptography aims to find secure algorithms to use in communication and is primarily theoretical work. The practical part is the translation of these algorithms to a program that can run on a computer, which is also a vital part of cryptography. In addition, both theoretical and practical work is done to try and probe systems and algorithms for vulnerabilities. This thesis is placed in the latter category and will investigate the practical application of an asymmetric cryptography algorithm called RSA.

When using the internet today we are typically use a browser or an app to communicate with some entity. The entity is often a server, which delivers some services of a company, such as movies or articles. Most of this traffic uses encryption and in this thesis we will look closer at some of the crucial aspects that allows this to happen securely.

Today we use mainly a combination of symmetric and asymmetric cryptography as a system. Chapter 2 will look closer at a protocol using this kind of combination .

Chapter 2

Certificates

Public key cryptography has one shortcoming; the man in the middle attack. If two users exchange public keys, how can they be sure that the key they receive is from the other user?

2.1 The man in the middle attack

The man in the middle attack (MitM) is when an attacker controls the communication channel we are using. The attacker can then read, write and manipulate the messages we send and receive. As mentioned, to communicate using public-key cryptography, you need to publish your public key. Doing so makes it possible for anyone to retrieve your key and send you encrypted messages. The problem is, as seen in Figure 2.1, that you can not know for sure who sent you the message. The attacker has a wide range

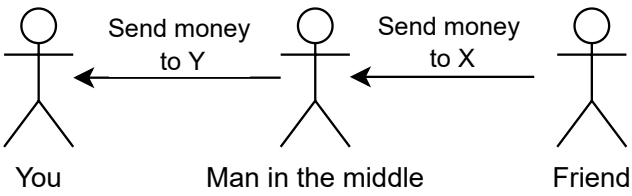


Figure 2.1: Man in The Middle attack between you and your friend

of possible malicious acts in this scenario. They can be passive by decrypting, reading and then encrypting the messages between you and your friend. Alternatively, they can be active by replace parts of or the entire message you and your friend send, as seen in Figure 2.1.

In Figure 2.2 we see how this can happen using public keys. When sending your key to your friend, the attacker intercepts it and sends his key to your friend instead. The attacker does the same when your friend sends his key, so you receive the attacker's key. Now the attacker controls the communication and the encryption. You and your friend think you are talking to each other, but all messages go through the attacker who can decrypt, read, modify and encrypt all messages before forwarding them. To solve this problem, the parties need some amount of trust between them. The parties involved need to be authenticated to ensure they are whom they claim to be. As we will see, this authentication is the main problem with public-key cryptography.

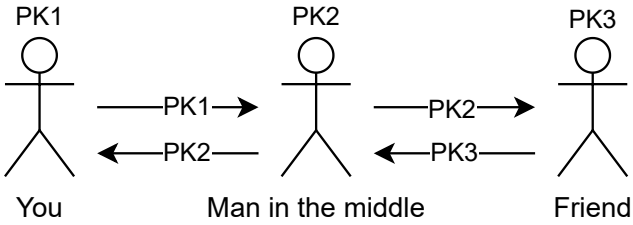


Figure 2.2: Man in The Middle attack between you and your friend, here we see how the MiTM attack works using public keys (PK)

Another consideration is knowing that the key your friend sends actually belongs to him. It can still be someone else using his computer or, in some other way, have access to his public and private key. Therefore, you only authenticate that the one you are talking to has access to the private key corresponding to the public key you have received. There needs to be something that binds your friend's identity to his public key, and this is done by using public-key certificates.

2.2 X.509 Certificates

A public-key certificate simply refers to a public key connected to some unique information about the owner's identity that is digitally signed. The unique information could be a person's name, name of a website or something else that is unique to the entity. This binds your friend's name to a specific public key, similar to the way an entry in a phone book binds your name to your phone number.

Today the most used standard for digital certificates is the X.509 standard. This standard comes from one of the world's oldest international communications organizations [23], the International Telecommunications Union (ITU), who in 1988 approved a standard named X.500, also called the Directory [24]. The X.500 was an international standard for directory systems, where the X.509 standard for digital certificates was named as the framework for authentication. The X.509 standard describes the format of public-key certificates, which, since then, has been updated and is widely used today.

Today we mostly use X.509 version 3. The format has remained principally the same, but some fields have been added over the years. In version 2, they added two optional fields for identifiers for the subject (the one receiving the certificate) and issuer (the one creating the certificate). The most significant change came in version 3, where they added extensions, which anyone can create and use, giving the standard flexibility. There are also some defined standard extensions, such as key usage. The key usage extension allows certificates to be used for specific purposes such as only for encryption, key derivation, authentication etc. The format of X.509 certificate version 3 has these fields in addition to the signature [9]:

1. Version Number- The version of X.509 this certificate is using (1-3), this is mostly version 3

2. Serial Number - The unique identifier for this certificate (Unique to the one creating it)

3. Signature Algorithm Identifier - The algorithm which was used to sign this certificate
4. Issuer Name - The name and information about the entity that issued this certificate
5. Validity Period - The period this certificate is valid, given by a start and end time
6. Subject Name - The name of the subject for this certificate
7. Subject Public Key Information - The public key of the subject and the encryption algorithm where the key should be used.
8. Extensions - optional extensions

These certificates binds the identity of an entity to a public key. The certificate itself should be signed in order to verify that the certificate is genuine. Signing, as already mentioned, verifies the integrity of a document and the authenticity of the sender. Back to the example of a certificate resembling an entry in a phone book, the question now is who maintains the phone book?

2.3 Public Key Infrastructure

> A PKI is the basis of pervasive security infrastructure whose services are implemented and delivered using public-key concepts and techniques [3]

A PKI is an infrastructure that uses public-key concepts to manage and control certificates. PKI is the infrastructure to maintain the phone book of public keys and identities securely.

The main tasks at hand for a PKI is certificate issuance and revocation as well as key management. Certificate issuance is the act of creating and giving a new certificate to an entity. Before issuing a certificate, the identity and credentials of the entity must be verified by some means. A PKI also needs to be capable of the opposite, namely certificate revocation. This is when a certificate is, for some reason, revoked before its expiration date. There are several reasons for this, such as the private key being stolen, lost or used fraudulently. A PKI must also manage the keys, such as private keys of users. The PKI manages the keys to be able to backup or recover lost or stolen keys. It can also be updating keys for security purposes [33]. This should be done in such a way that the one managing the private keys cannot read or use them in any way as this would defeat the purpose of private keys.

Today there are several PKI models based on different trust models we will look at some of them.

2.3.1 Certificate Authorities

Today the most used PKI model is based on Certificate Authorities (CA). The model is based on trusting a third party known as a CA. In PKIs using CA, the trust is a hierarchical model in where a root CA sits at the top. A CA is responsible for binding

the identity of an entity to a public key. This means that they need to vouch for binding an entity's identity to a public key in the form of signing their certificate [3]. A CA essentially manages certificates. They create, issue and revoke certificates. Today there are hundreds of CAs worldwide, and as a whole, they follow a hierarchical model of trust.

When you need a certificate today, you can ask a CA for one. The CA will then do some kind of verification of your identity. For instance, as a business or a domain, they will ask for some document proving you are the owner of the business or domain. There are some standards for this verification, Organisation Validation (OV), Domain Validation (DV) and Extended Validation (EV) where EV is the most comprehensive [7]. Then they will create a certificate, sign it with their private key and issue it to you. When you want to communicate with your friend again, you can send your certificate. Your friend can then retrieve the CA's public key to verify the signature on the certificate. Now, as long as he trusts the CA, he knows he is talking to you as long as he uses the public key stated on the certificate. However, how does your friend know he has the CA's original public key?

The CA's certificate is signed by another CA, the same way the original CA signed your certificate. This forms a chain of certificates. At the end of this chain is a top CA,

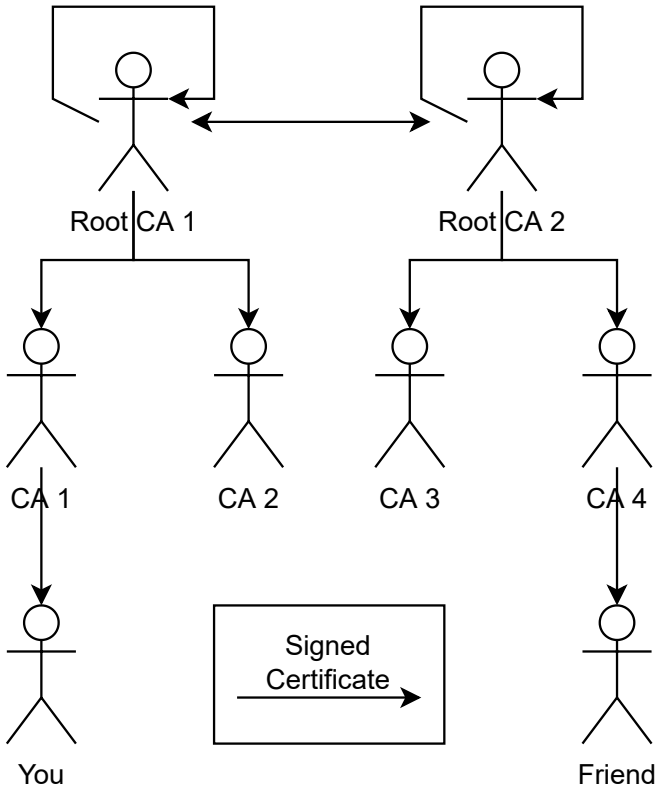


Figure 2.3: Hierarchical CA model. To be able to verify your friend's certificate there needs to be a path between you. Here we can see the root CAs signing each others certificates, known as cross-certification.

also known as a root CA, as seen in Figure 2.3. The root CA has a self-signed certificate that its public key can verify, this is called a trust-anchor. This act of verification of a root CA is, in the end, no different than your friend self-signing his certificate. The

difference is that a root CA is a business that relies on trust to survive and should be audited frequently. When you send your certificate to your friend, it is necessary to also send the certificate chain. This is because your friend might not know the CA you have used, or, more importantly, does not have their certificate. So when you send the whole chain, he can then verify each certificate and its signature up to a root CA. Then the requirement is simply that your friend has a way to get the root CA's certificate. This model has several variations, such as a more strict model where there is only one root CA that you and your friend trust.

2.3.2 Web of Trust

The web of trust model is based on each user being its own CA. The model was published and used in Pretty Good Privacy (PGP) by its creator Phil Zimmermann in version 2.0 in 1992 [36]. PGP is an email program where the users create certificates linked to their email addresses. This way, every user can issue certificates for themselves. The users sign their certificates and send them to their friends, who also can sign the user's certificate, as seen in Figure 2.4. Their signatures are appended to the certificate. This way, a certificate can get many signatures which help to verify the certificate's authenticity. The user has good control over the protocol and can decide the level of trust for other users. [33].

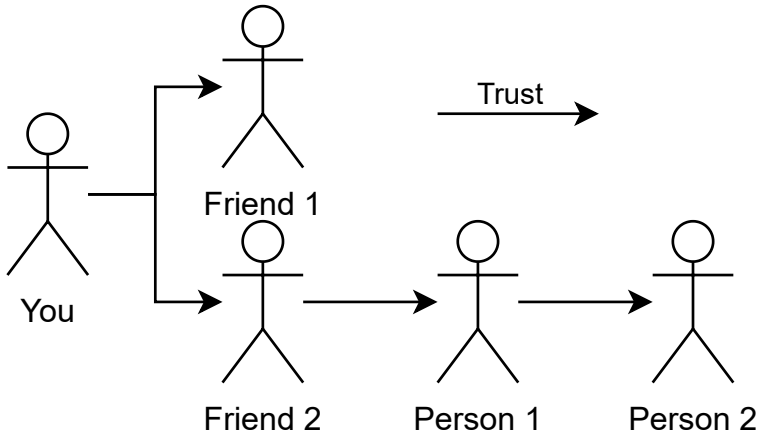


Figure 2.4: Web of Trust, Relies on implicit trust between parties. The figure shows how using web of trust you can implicitly trust someone you personally don't know such as person 2.

As users gather certificates, they are added to a list with two values associated with each user and their certificate. These values are the levels of trust associated with a certificate, where there are two central values. The first is called the "key legitimacy field" and is the trust in the user and its keys validity. The second is called the "owner trust field" and is the trust of the user to sign other certificates. Together these fields determine to which degree the user trusts other users.

There are degrees of trust, so you do not necessarily trust everyone your friend trusts. As you have the two fields associated with your friend, it is possible to trust him, but not trust his friends. There are some drawbacks of this model, one being its scalability. This model suits small groups where people know each other. It is harder for larger groups or companies since not everyone knows each other, and it is

also difficult to enforce uniform policies. Another drawback is the level of technical understanding required to use it. Users must create and sign certificates, maintain the list of certificates, as well as how much trust they have in them [33].

2.3.3 Web PKI model

When browsing the web today, we primarily use the web PKI model, which in essence is a list of root CAs that comes with our browser [3]. As long as the website we visit has a certificate that can be validated up to a root CA from this list, the certificate is considered genuine. This is only true as long as all signatures are verified and all certificates are in their validity periods. The independent root CAs can be seen as a strict hierarchical model, and the browser trusts all the root CAs. Today this list consists of around 150 CAs, some more or less known. Some devices also come with such a list, which some browsers such as Chrome use instead of having their own list.

The advantages of this model are its simplicity and its ability to scale well. The model also has some drawbacks. The main issue is that someone else is deciding whom you should trust. The user can change this list, but most users do not know how or what to change. Another drawback is the equal trust the CAs have. For example, if one CA would issue a certificate to someone malicious for the Google domain, the browser would see this as valid. The point here is that the model trusts each CA equally and thus is only as strong as its weakest link. To continue to the next drawback, say a CA was bad or had its keys lost or stolen. How would this be handled? The CA should obviously no longer be trusted, but there is no complete and general way of distrusting or revoking the CAs in the web model.

For example, this has occurred in the past with the CA Diginotar, which was hacked in 2011. The attack led to rogue certificates being issued by "Diginotar" and trusted by web browsers worldwide. The attack used these rogue certificates in a large scale MiTM attack on around 300,000 users. The investigation [22] showed that there were shortcomings in the security at Diginotar. "Diginotar's" false certificates were used for almost two months. So what happens in cases like this? It is up to the browsers to remove the root certificates and publish an update. In Diginotar's case, the major browsers distrusted Diginotar's certificates, and Diginotar declared bankruptcy a few months later. As a curiosity, this event was what inspired the creation of the CT logs which we talk about in Chapter 4.

Another thing to point out is the terms and conditions, and the responsibility between a user and a CA. Consider a situation where you only trust one CA out of hundreds. Then you can use their services and agree to some terms and conditions that outline both parties' responsibilities. This gives you hundreds of options for terms and conditions as you can choose any of the CAs. The web PKI model removes this agreement or at least the opportunity of this agreement between a CA and you. Instead, it is replaced by a browser or large corporations, such as Apple and Google, which uses hundreds of CAs and limit the number of terms and conditions.

The Web PKI model has changed somewhat since its beginning, for example, by the Certificate Transparency (CT) initiative by Google. This initiative gives some transparency to the web PKI model by encouraging certificates to be added to logs open for anyone to read and verify. The encouragement lies in treating websites that have

not uploaded their certificate as less secure by the browser [14]. As each certificate's certificate chain is uploaded, this also gives insight into who issued and signed the certificates. We will come back to the CT logs in Chapter 4.

2.4 Secure communication between webservers

When we are browsing the internet, we usually communicate with some server somewhere. One of the most widely used protocols to achieve this securely, is the Transport Layer Security (TLS) protocol using the described web PKI model. TLS is used in most internet traffic, as it ensures authentication via the web PKI model and integrity and confidentiality of the communication using public key and symmetric cryptography.

2.4.1 The TLS protocol

The TLS protocol is a protocol to achieve secure communication when using the Transmission Control Protocol (TCP). This is done by a combination of cryptographic tools and a PKI. The protocol works by defining a specific way to communicate between a server and a client.

TLS protocol was first launched as Secure Sockets Layer (SSL) by Netscape, the popular internet browser, in 1995. This was the second version, as the first version was never published because of severe security flaws. The SSL protocol was widely used and, as follows, thoroughly investigated. This resulted in new updates with better security. The protocol was standardized by the Internet Engineering Task Force (IETF) as TLS, a slight modification of SSL version 3 [26]. The protocol follows some steps

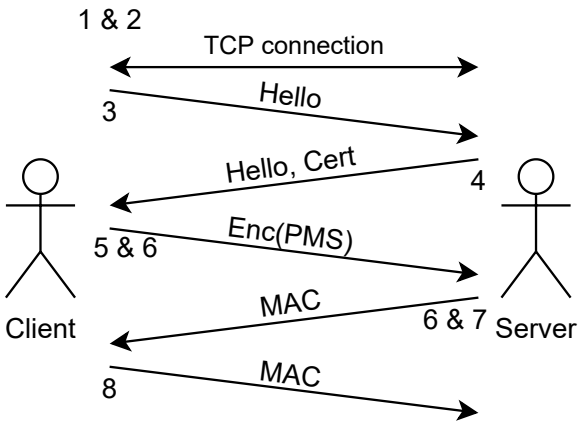


Figure 2.5: TLS 1.2 Rough outline of TLS setup.

to set up secure communication between two parties. The messages of TLS version 1.2 are shown in Figure 2.5.

1. It starts by setting up a TCP connection. TCP is the protocol used to send and receive unencrypted messages. This starts with the client sending a message to the server.

2. The server responds and sets up the TCP connection. Now the client needs to respond to use the connection.
3. Then, the client sends a list of cryptographic algorithms available along with a random number used only once, called a nonce to the server
4. The server picks the best cryptographic algorithms and sends this along with its public key certificate and a nonce to the client
5. The client can now authenticate the server by checking the certificate and its chain up to a root CA. Then the client extracts the public key and generates a Pre-Master Secret (PMS), encrypts it using the public key and sends it to the server
6. The server can now decrypt the message to get the PMS, and both client and server use a Key Derivation Function (KDF), a function to generate uniformly random keys, to get the same Master Secret. They also generate some other keys used for MAC
7. The client sends a MAC of all the previous messages
8. The server sends a MAC of all the previous messages. The fact that the server and client share the same MAC key proves that the server must have the private key associated with the certificate. This authenticates that the client is talking to the server the certificate is issued to.

This is not all the complete details of the TLS protocol but a good overview [11]. As stated, the authentication comes from verifying the certificate and the server being able to finish the handshake protocol successfully. The confidentiality comes from the use of encryption, both symmetric and asymmetric. The integrity comes from the use of MAC to ensure that no messages have been changed. The MAC also provides authentication, but since it relies on the shared key, it relies on the former authentication from the certificate. When these steps are done, the parties have a shared key they can use to encrypt their messages with the agreed-upon symmetric encryption algorithm. So if chosen, RSA can be used in two ways in TLS. It can provide authentication, as a algorithm used for the signature on the certificate. It can also provide confidentiality, as a encryption algorithm for the PMS.

For this thesis, it is important to notice the usage of certificates. When starting a TLS connection, a certificate is used, and as we will see later, one way of collecting RSA keys is to initiate a TLS connection and retrieve the certificate presented. One can also do this on a broader scale by asking every IP address to set up a TLS connection and collect the certificates presented. When asking for every IP address, this is called an internet-wide scan.

Another thing to notice is the impact it would have if the RSA key from the certificate was used to encrypt the PMS, and that key was broken by someone malicious. The malicious actor could then create the keys used to encrypt the communication, breaking the encryption. In this case, both confidentiality and authentication are broken. This is also true for messages between other parties that communicate with the server if the same encryption method and the same key were used.

It is also possible to decrypt old messages. If the malicious actor recorded the encrypted communication, the actor could break the key and decrypt the messages at some

later point in time. Forward secrecy prevents this and is one of several improvements in TLS 1.3 [31].

Forward secrecy is when encrypted communication cannot be decrypted, even if the private key used is compromised at some later point in time. This ensures that even if someone were to log all your internet traffic, they would not be able to decrypt all previous messages even by breaking the private keys used. One way of accomplishing this is by using session keys. A session key is only used for one session, which could be one message or a hundred, but the principle is that the key changes. This limits the impact of cracking such a key to the specific session. The main idea is that it should be impossible to get from one session key to the previous.

Chapter 3

RSA and factoring

3.1 The RSA cryptosystem

In 1978 Rivest, Shamir, and Adleman discovered the first practical public-key encryption and signature scheme, referred to as RSA. The RSA scheme is based on a hard mathematical problem, the intractability of factoring large integers [4]. The public-key scheme's goal is to establish a shared secret while communicating on an open channel, and the RSA is one of these systems most frequently applied. In general these systems were revolutionary and would end the ever-growing hassle of key management of symmetric encryption algorithms. In RSA, this is done by exchanging integers in a specific way.

To generate a RSA key one should generate two distinct prime numbers of similar sizes, p and q (factors), and calculate their product n (modulus).

$$p \cdot q = n \tag{3.1}$$

Next, choose the encryption exponent e which is often just a standard exponent. Finally, compute the decryption exponent d by using Euler's totient function. As Euler's totient function is very easy to calculate, if you know the factors of n, and very hard if you don't, we can see how it is infeasible to find d without knowing Euler's totient function of n. This is again only possible if we know the factors p and q of n.

$$\phi(n) = (p-1)(q-1) \tag{3.2}$$

$$d \equiv e^{-1} \bmod \phi(n) \tag{3.3}$$

Now we can see the relationship between e and d as

$$e \cdot d \equiv 1 \bmod \phi(n) \tag{3.4}$$

In practical application, this is usually done differently to speed up the process of decryption. The public key consists of the modulus n and the encryption exponent e. The private key is the prime numbers p and q and the decryption exponent d. Now you can send someone your public key and they could encrypt some message m by computing

$$c \equiv m^e \bmod n \tag{3.5}$$

and you could decrypt it by computing

$$m \equiv c^d \bmod n \tag{3.6}$$

We can see that this works as

$$(m^e)^d \equiv m^{e \cdot d} = m \bmod n \tag{3.7}$$

The drawback of RSA is the speed at which one can encrypt and decrypt [33]. It is mainly used in the early stages of communications to exchange symmetric encryption keys in modern applications. This is, for instance, done by TLS as described in Chapter 2. RSA is one of the asymmetric encryption algorithms widely used for symmetric key exchange.

3.1.1 Security in RSA

The security in RSA lies in the hardness of factoring large numbers. This is the main security aspect, but there are other practical security aspects in applying RSA, as with any cryptosystem. These can be parameter selection and how the system is used in practice. Some examples of choices that could undermine the security are choosing too small primes p and q or incorrect use of padding. The RSA system is dependent on using a suitable padding scheme, which adds randomness to the message to avoid leaking information about the message being sent. Another important part is choosing the p and q so that it is infeasible to factor n. The primes should be large and about the same bit-length, and the difference between them should not be too small. If the difference is too small, then one could factor n simply by applying the square root and looking at the primes close to it [4]. The reason for choosing primes that are roughly the same bit length is to not be easily factored by the elliptic curve factoring algorithm, which factors "small factors" fast. Today the size of p and q should at minimum be 1024-bit or larger to be secure. That makes the modulus n a 2048-bit number.

3.1.2 Factoring

If one could factor a 2048-bit integer quickly, RSA would be broken. Fortunately, this is a hard problem that has been well studied, and so far, the largest number factored that we know of is an 829-bit number [37]. The factorization problem is to factor a large integer n into two or more distinct prime numbers on the form $n = p_1^{e_1} \cdot p_2^{e_2} \ldots p_k^{e_k}$. There are several algorithms to factor large integers. Some are specialised, while others are general. The specialised algorithms take advantage of the factors being in some special form. These generally perform better than the general-purpose algorithms but, in return, can only factor some numbers. Some examples of special-purpose factoring algorithms are trial division, Pollard's rho algorithm and Pollard's $p-1$ algorithm. Some general-purpose factoring algorithms are the quadratic sieve, and the general number field sieve algorithms [4]. These algorithms are used to factor large integers, and as the computers are getting faster and faster, the size of the numbers we can factor is also growing. For example, the record mentioned above took more than 2700 core years and used the number field sieve algorithm.

3.2 Greatest common divisor and previous research

So factoring RSA keys is very difficult, but there are other threats to the security of RSA. We are trying to test one of them, leveraging the fact that hundreds of millions of RSA keys are publicly available. Because, what happens if two different RSA keys share one factor?

In this case, it would be trivial to factor them by using the greatest common divisor (GCD) algorithm. The GCD algorithm is an algorithm that, in our case, only takes two numbers as input and outputs their greatest common divisor. So if we take the GCD of 15, whose factors are $3*5=15$, and $25=5*5$ it would output 5, and if we used 15 and 4 it would output 1. An important note here is that since we work with RSA keys n which are products of two primes p and q, the only factors of n are n, p, q, and 1. Therefore, the GCD of two RSA keys that share a factor would return the shared factor breaking both keys.

Today, the most used key size in RSA is 2048-bit, that is a 616-digit number represented in decimals. This means the prime numbers should be around 1024-bit in size or a 308-digit number in decimal. Even though we only choose prime numbers from this space, there are still more prime numbers than RSA keys in the world by far. In the example of a 2048-bit RSA key, the primes of 1024-bit are then 308-digit numbers. An approximation of how many primes one can choose from up to some x is $x/\ln(x)$. Using this approximation and calculating the difference between $x=10^{308}$ and $x=10^{307}$, we find approximately how many primes we can choose for our 1024-bit primes. With our example, this is roughly a 305-digit number. For comparison, an estimate of how many atoms there are in the universe is 10^{84} an 85-digit number [5].

So, choosing the same prime twice should not be an issue, as the space to choose prime numbers from is so large. Such a collision of numbers would be a miracle in a perfect world. However, this is only the case if the prime numbers are actually picked at random.

It turns out this was a severe problem discovered and disclosed by Heninger et al. [21] and Lenstra et al. [28]. Heninger et al. [21] performed an internet-wide scan on ports 443 (HTTPS) and 22 (SSH) to collect RSA keys and found 5.6 million and 3.8 million distinct RSA public keys, respectively. They also collected 3.9 million keys from the SSL Observatory [13] totalling at 11 million distinct public RSA keys. They factored 16,717 of these 11 million keys by taking the *GCD* between all pairs of the RSA moduli.

Since then, the same experiment was done in 2016 by Hastings et al. [20] which collected 81 million distinct RSA keys and factored over 313,000 of these. In 2019 Kilgallin et al. [25] collected around 57 million distinct RSA keys from an internet-wide scan and was able to factor 249,548 of these keys. They attribute the high number of broken keys to the rise of the Internet of Things (IoT), as IoT devices are primarily lightweight and might have a poor entropy source. For comparison, Kilgallin et al. also collected 100 million keys from the CT logs. For this data set, they were only able to factor 5 keys.

The difference between the sources of the collections of RSA keys is essential. The CT logs are certificates uploaded mostly by CAs, while an internet-wide scan might return TLS certificates not created by real CAs. A TLS certificate returned from such

a search might be in use or be created for testing purposes. The point here is that there could be a difference in the quality of certificates found in CT logs and the TLS certificates from a internet-wide scan. Another way of looking at it is that the TLS certificates may be self-signed and used for personal use or only within organisations. However, to have your certificate in a CT log, it must be signed by a CA trusted by major browsers. The CA may also use their own software to generate the keys and therefore, have more control over how you generate your keys. As the CA's business model is built on trust and security, they should be good at creating secure certificates and strong keys.

Author	Year	# of distinct RSA keys	# of factored keys	Percentage
Lenstra et al. [28]	2012	11 million	16,717	0.152 %
Henninger et al. [21]	2012	6 million	12,934	0.215 %
Hastings et al. [20]	2016	81 million	313,000	0.386 %
Kilgallin et al. [25]	2019	57 million	249,548	0.438 %
Kilgallin et al. *	2019	100 million	5	0.000005%

*Table 3.1: Results from previous research. Here we can see the number of collected RSA keys and how many of them were factored. *The 100 million RSA keys were collected from the CT logs.*

It seems that the results of factoring depends on where the keys are collected from. All previous studies mentioned have performed some degree of internet-wide scan and this is where most keys are factored. Only the study from 2019 tries to factor keys from the CT logs. A summary of previous research is given in Table 3.1 As of 2012, the standard was 1024-bit keys, and now the standard is 2048-bit. The majority of RSA keys found in the CT logs are 2048-bits. It is unclear what the key size was in the past experiments, but most of them likely included key sizes of 1024-bit. We wanted to replicate the GCD experiments done in 2012, 2016 and 2019, but we are exclusively looking at 2048-bit keys.

3.3 Random number generators

To generate an RSA key, you need two random prime numbers, and these are critical for the security of RSA. There are two types of random generators, a pseudorandom number generator (PRNG) and a hardware random number generator (HRNG). The PRNG is essentially an algorithm that quickly can generate large amounts of random numbers given a seed. The seed initiates the generated sequence, so using the same seed will generate the same sequence of random numbers. This is why the HRNG is also used. HRNG is a slower generator but is supposed to be truly random. So using the HRNG to seed the PRNG makes a truly random generator capable of quickly generating large sequences of random numbers [33].

As already mentioned, Heninger et al. found some serious issues here. For example, in Linux, when booting there was an entropy hole causing the random generators to not generate random numbers, leading to the PRNG being essentially deterministic. This would lead to RSA keys generated in this state being the same worldwide. Another finding was that most of the weak keys were generated in headless or embedded devices. This is cause for concern, as the number of such devices increases faster than

ever. As stated, Kilgallin et al. [25] factored many keys which they attribute to IoT devices from their internet scan. From their CT log collection, they factored five keys. So finding such weak keys which share one factor with another key could be a symptom of a not so random PRNG or HRNG.

3.4 Factoring RSA keys using greatest common divisor

3.4.1 GCD between all pairs

After collecting the large set of m RSA-moduli from the certificates, the analysis seems quite simple on the surface: check if any of the 2048-bit integers share a factor that is not 1. The naive solution is perhaps to simply run through all numbers and check the GCD of all pairs.

$$GCD(n_i, n_j), \text{ where } i < j < m \tag{3.8}$$

Nevertheless, this will have $O(m^2)$ runtime, which takes too long when m is large.

Another approach is to multiply all the numbers together, dividing by the first number and checking the GCD of the large product and the first number. Then iterating through all the numbers yields $O(m)$ runtime.

$$GCD(n_i, n_1 \cdot n_2 \cdot \ldots \cdot n_{i-1} \cdot n_{i+1} \cdot \ldots \cdot n_m), \text{ where } 1 \leq i \leq m \tag{3.9}$$

The problem with this approach is that the product will be very large in terms of memory, and the operations, including this product, will be very slow.

To resolve both these issues, Heninger et al. [21] implemented an advanced GCD algorithm. This algorithm tries to limit the number of operations on the large product and use memory efficiently. The algorithm is based on the same approach as above, except that instead of using the large product N in m operations, it uses a product and remainder tree as seen in Figure 3.1.

Here we see the algorithm's inner workings, namely the product tree and the remainder tree. The algorithm is built on the following equation showing how we can calculate the GCD of a large product and one of its factors.

$$GCD(n_i, n_1 \cdot n_2 \cdot \ldots \cdot n_{i-1} \cdot n_{i+1} \cdot \ldots \cdot n_m) = GCD(n_i, \frac{n_1 \ldots n_m \bmod n_i^2}{n_i}) \tag{3.10}$$

This is proven by Bernstein [6], the fundamental discovery is that the GCD operation is equally valid if done on the fraction in Equation 3.10. This allows us to do the GCD with far smaller numbers than the entire product, allowing us to use large inputs. The tree structures are meant to reduce the amount of memory the algorithm needs to keep in RAM at any given moment and reduce the number of operations on the large products. This is done by building the structure in Figure 3.1 line by line from top to bottom. The product tree is built by multiplying numbers as a binary tree, and the remainder tree is the large product mod each node from the product tree squared.

In the product tree, we only keep the previous level in memory while calculating the next level. While in the remainder tree, we need both the previous level and the equivalent level in the product tree, so this is the most memory demanding process. We can also see from the figure that the number of operations with the large product

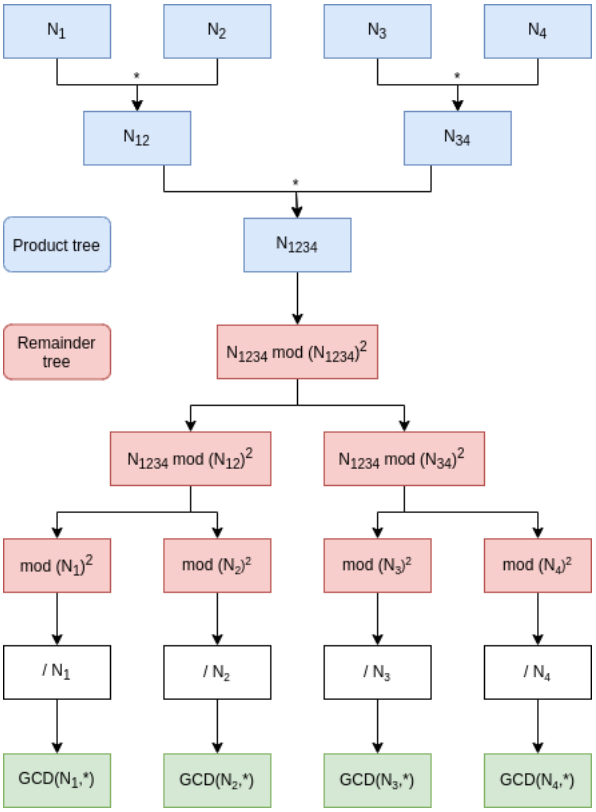


Figure 3.1: Product and remainder tree in GCD Algorithm

is reduced to two operations. We also note that in each level in the remainder tree, the bit-length of the input is half of the inputs on the level above. So the bottleneck is at the top nodes in the remainder tree. Here all the integers are huge and require a lot of processing.

This is the algorithm Heninger et al. wrote in [21] but was updated by Hastings et al. [20] to run on a distributed system in order to avoid the bottleneck at the top nodes in the remainder tree. They called it the batch GCD algorithm.

3.4.2 Batch GCD algorithm

The batch GCD algorithm is similar to the algorithm described above, but instead of having one product and remainder tree, it has several. The fundamental principle of the algorithm is to split the initial list of keys into some K batches, then run the algorithm on each batch as seen in Figure 3.2. Simply running the algorithm would only find shared factors within each batch. In order to check if keys from different batches share a factor, we need to run the algorithm K^2 times because each batch's large product needs to be checked against all other remainder trees. This increases the runtime in theory, but it makes the algorithm faster in practice. This is by controlling how large the product at the root of the remainder tree can be, thereby controlling the scale of the bottleneck. This also allows us to handle bigger sets of keys since each batch is smaller in RAM than the whole set would be.

It is also worth mentioning that when checking a product from one batch against a remainder tree from another, one should not divide by the modulus before doing the

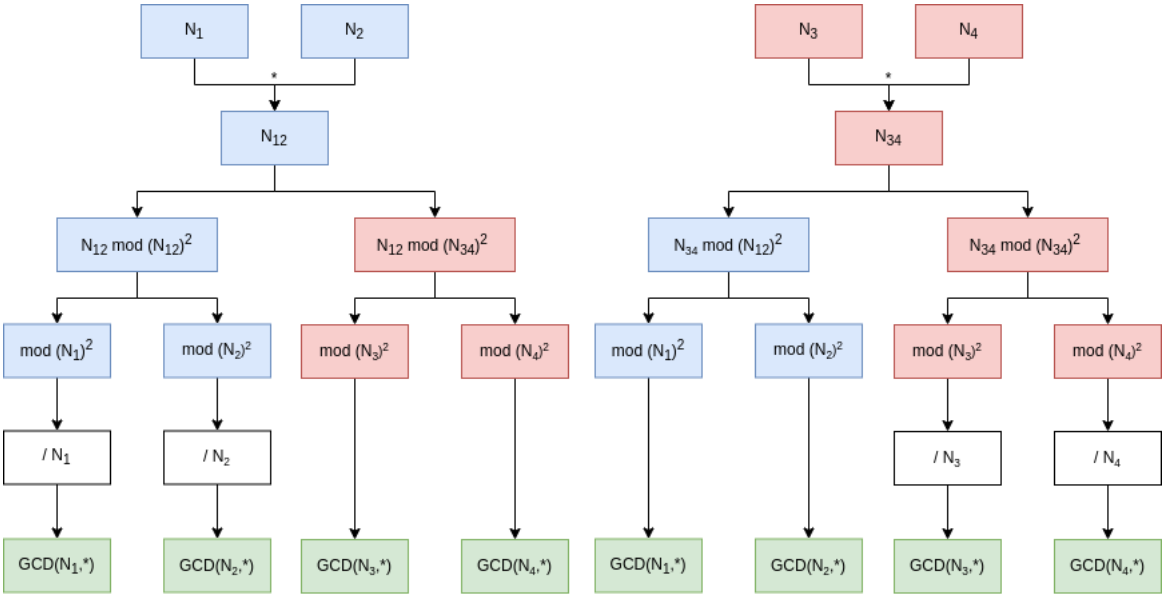


Figure 3.2: Product and remainder trees in Batch GCD Algorithm: The batch algorithm is $O(K^2 m log(m))$ since each batch needs to be checked against all other batches. Here we also note the difference between checking "inside" a batch and checking across batches. In our implementation, the large products N_{12} and every N_i^2 would be written to file

GCD at the bottom of the remainder tree. This is because the dividing step is partly to remove the entire key, so it is not the result of the GCD. However, using a product and a remainder tree from different batches would result in not finding the shared factor. The algorithm we wrote is based on the one from Heninger et al. [21] and Kilgallin et al. [25] but with some minor modifications. Both our and Heninger et al. algorithms write each level in both the product and remainder tree to files. We used OpenMP [2], and instead of writing all levels in the product tree to file, we wrote every other level, as suggested by Kilgallin et al. Another change we made was calculating the squares of the products in the product tree method instead of the remainder tree method. Here we also wrote the product tree nodes squared to file as they are larger in size than the remainders in the remainder tree. So when running the algorithm, what is calculated in each run is the product N modulus of the product tree squared N_i^2. The value of N_i^2 is simply read from the file.

3.4.3 Optimizations

We also planned on not doing mod N_i^2 but mod N_i when checking a product against a remainder tree from a different batch. Doing this would decrease the overall run time and use less memory but double the number of files written. In our approach, this would mean writing every other level in the product tree to file, both squared and not squared. In total, this would increase the memory usage by around 50%. When considering the overall runs, the minority are done "within" the batches, and the majority are across batches, so improving these runs would have the most impact.

Chapter 4

Running the experiment in practice

This section will describe how we did the project in practice and cover the results.

4.1 Certificate Transparency logs

The CT logs are an initiative started by Google, that aims to provide more security and transparency to the web PKI worldwide, by hosting servers from Google and other companies that store TLS certificates. The main feature this provides is that anyone could upload their certificates, however, most are uploaded by CAs. The logs can then be verified and investigated by anyone. The CT logs are also created so that one can only append new certificates and read existing certificates, stopping anyone from tampering with the logs. Google also implemented Chrome in 2018 [14] so that a website that has not uploaded its certificate will be treated as less secure than one that has uploaded its certificate to the CT logs. This made it even more used, which led to the logs vast number of certificates. When this is written, the logs contain over 6 billion certificates stored in around 40 different logs [1].

We collected 159 million certificates which used 2048-bit RSA keys for our analysis. This was the most time-consuming process in our experiment.

The majority of the logs are hosted by Google, which also holds the majority of the certificates on their logs. All logs limit the maximum amount of certificates received per request ranging from 32 to 1024 certificates. We wrote a script in Python inspired by Sears [32] to collect certificates from one of Google's logs named Argon 2021 [8]. This script would retrieve some certificates, check if they met our requirements, and then write them to a file with their unique (for each log) index.

Our first requirement was that the certificates had actually been used and were not pre-certificates, as some logs hold more pre-certificates than regular certificates. Pre-certificates are essentially also X.509 certificates but cannot be used as one. The primary use of pre-certificates is to be uploaded to a CT log to retrieve a valid signed certificate timestamp that can be added to the actual certificate. This makes it possible to gather the relevant information in the actual certificate.

The next requirement was that the certificate contained a public RSA key of size 2048 bit. This filtering also elongated the time it took to obtain the certificates since not all certificates met our requirements. Using these requirements to gather our 159 million certificates led us to check over 400 million certificates from Argon 2021.

The log we chose had a limit of serving at most 32 certificates per request, and we got around 150,000 certificates per hour per CPU. We used 23 CPUs in parallel, giving an overall runtime for collecting 159 million certificates of around 48 hours. There are ways this can be done somewhat faster using other tools and methods [32]. When collecting all the certificates, we needed to extract the public key from the certificate and write it to a file in a specific format for our algorithm. As we had collected the certificates and created one large file for our algorithm, the final step was to remove all duplicates from the file.

4.2 Running the batch GCD algorithm

When running the batch GCD algorithm, we also encountered the same bug in the GMP library [19] as Heninger et al. did in their implementation [21]. The bug happens when writing large numbers to a file, since the maximum output size GMP will write to file is 2^{32} bits, equal to 4 GB. The squared product in our remainder tree exceeded this limit. We only first encountered this when actually running the algorithm on our data set. The bug is in the function mpz_out_raw, and we used Heninger et al.'s patch to proceed.

We split our input of 159,377,664 million RSA keys into $K = 13$ batches making each batch holding around 12 million RSA keys. Each level in the product tree took 3 GB, and the same levels squared used 6 GB of disk memory. With the depth of each tree being around 24, this would be 144 GB per tree, and with 13 such trees, it would require around 1.8 TB of disk memory. To half this memory usage, we only stored every other level of the trees as mentioned in Section 3.4.2. Our algorithm ran in 88 wall clock hours on 30 cores using 180 GB RAM at the maximum. It also used around 1 TB of disk memory for the files it wrote. Checking one product against another batch's remainder tree took around 30 minutes, and with $K = 13$ we had 169 such "runs". We also added some dummy keys with shared factors to the data set to verify that the algorithm actually would find them. If they were not found, it would indicate a programming error.

4.3 Results

The algorithm found eight keys with shared factors out of the 159 million, in addition to our dummy keys. We verified that the found factors were indeed primes and that their products multiplied to the moduli given on the certificates. All the certificates expired in early 2021, and as stated earlier, all keys were 2048-bit RSA keys. Upon closer inspection of the broken RSA keys, we noticed that their certificates all were issued by a CA named ZeroSSL. Another peculiar finding was that the domains on the certificates were all Vietnamese. Other than this, we could not find any apparent connection between the keys.

This result would in itself be pretty close to what Killigan et al. [25] found in 2019 when checking 100 million RSA keys from the CT logs, as they only factored five. This, combined with our results, could suggest that around 0.000005% of the CT logs are vulnerable to this method. As Killigan et al. did not reveal the source of the vulnerable certificates, we cannot say if it's the same as we found. Either way, this is a low but still significant number of vulnerable certificates.

4.4 ZeroSSL

ZeroSSL is a certificate authority with its main office in Vienna and a branch in London. They have been in business since 2016 and now have more than 500,000 customers worldwide [35].

After our initial results, we decided to continue to investigate why we only found weak keys from ZeroSSL and wondered if there were more. We started a new collection, going through the entire Argon2021 database and only storing certificates issued by ZeroSSL containing 2048-bit RSA keys. As Argon contains over 1,3 billion certificates, it took us around ten days using 25 CPUs. This resulted in a data set of 716,228 such certificates. We then ran the GCD algorithm once more on this data set and found 355 keys we were able to factor. Once more, most of the domains associated with the certificates were Vietnamese, and those not under the .vn domain had Vietnamese sounding names. Other than that, we did not find any connections between the certificates.

We used an online classification tool developed by Svenda et al. [34] on a subset of the 355 keys, which classifies which library is most likely to have generated the RSA keys. This is possible due to bias when choosing the prime numbers for the RSA key. These results are not necessarily accurate as the classification tool needs large and updated samples from each library to classify it correctly. That being said, the results with 100% accuracy were that one of the following libraries generated these keys. This could indicate which library generated the vulnerable keys, although we only found ZeroSSL's keys, suggesting that it might be a local issue.

- Cryptix JCE 2005 03 28
- FlexiProvider
- mbedTLS <= 2.8.0
- Nettle <= 2.0
- PuTTY
- Sage
- SunRSASign

We reached out to ZeroSSL by email about the broken keys on their certificates. They quickly responded and wanted to know more about the vulnerability. We explained our experiment and our findings to them. They needed some of the certificates involved to verify the finding, and we sent them the factored keys and corresponding certificates. They started an investigation, and not long after, they responded with an explanation.

When a customer at ZeroSSL wants to generate a certificate, they have two choices on how to do so. The customer can either generate their own key and submit it, or generate a key using ZeroSSL's browser-based generator. In the case of the broken RSA keys, all the domains belonged to one single user who ZeroSSL says abused their systems by utilizing an automated browser. They concluded that this was most likely the cause of our findings. As the user used ZeroSSL's browser-based generator, such

an automated browser could be the reason for the lack of randomness when choosing the primes.

We can not verify that all the certificates belonged to the same person, but it can make sense considering that most of the affected domains are Vietnamese or have Vietnamese sounding names. Although the user used an automated browser to use ZeroSSL's services, in the end, it was ZeroSSL's browser-based generator that generated the bad keys. The user probably did not know that using ZeroSSL's service this way would lead to insecure key generation. In any case, the probable cause for the factored keys is still the underlying problem of lacking randomness.

Generally speaking, this is a serious security concern. In dealing with cryptography, it is crucial that random generators are tested thoroughly on what is being generated by it, as not implementing this properly could result in severe vulnerabilities.

4.5 Other observations

Here we discuss some peculiar things we found doing the experiment that we wanted to include in the thesis.

4.5.1 Exponents

During our initial collection, we also stored any encryption exponent e that was not 65537, which by far was the most chosen exponent. The choice of exponent has some conditions that need to be met and has a relation to the decryption exponent d as described in Chapter 3. Beyond this, the choice of e or d and the calculation of the other is individual. Primarily $e = 65537$ is used for efficiency reasons, and d is calculated based on the chosen n. Reasons for altering this could be to speed up the process of decryption or encryption by getting a low e or d. You can see the frequency of the other exponents we found in Table 4.1.

One of the most used non-standard exponents we found is 3. This should be secure to use, given that proper padding is used. Since padding is used by all serious libraries implementing RSA, this should be safe [30] [33]. PKCS#1 gives recommendations for implementing and using RSA and recommends using a suitable padding scheme such as Optimal Asymmetric Encryption Scheme (OAEP) [29].

We also note the possibility of some of these exponents leaking factors of $\lambda(n)$ described by Langkemper [27]. When generating e it needs to be coprime with $\lambda(n)$. As n is chosen, a naive implementation could be to start at $e = 65537$ and check if its coprime with e. If e is not coprime, one could increase e by two and check again. This would however reveal that $e = 65537$ is a factor of $\lambda(n)$. For example, when starting at $e = 65537$ and increasing by two and by chance ending up at 65541, we may expect that both 65537 and 65539 are factors of $\lambda(n)$. If this is the case, it leaks information on the factors of n.

Although not insecure, more careful consideration of how and why these exponents were chosen in RSA would be interesting.

Exponent	Frequency
35	39
3	37
65541	15
37	11
4097	7
7	4
5	3
64345	2
65569	2
17	1
34877	1
35697	1
38633	1
41749	1
52019	1
53827	1
56255	1
56527	1
57137	1
63515	1
63677	1

Table 4.1: Frequency of exponents used other than 65537, among 159 million different keys less than 200 had a different exponent. We list only the smallest exponents and the highest frequencies, the largest exponent we found was 130377.

4.5.2 Multi Domain Certificates

As mentioned in Chapter 2 X.509 has an extension called subject alternative name (SAN). The extension allows certificates to list multiple domains, emails, IP addresses, and other identities to be associated with the certificate. As stated in RFC 5080 [9]:

> Because the subject alternative name is considered to be definitively bound to the public key, all parts of the subject alternative name MUST be verified by the CA

So an user can request a certificate containing several domains, and as long as the CA verifies that the user controls these domains, the certificate can be issued. Such certificates that have several domains listed in the SAN extension are called Multi-domain certificates [12]. We found some of these when working with the certificates from the CT log.

One criteria for the batch GCD algorithm is that each RSA key is unique. That is, there should be no duplicates when running the algorithm. This ensures that the factors are found instead of n being returned as the GCD. We, therefore, needed to find and remove all the duplicates in the 159 million keys.

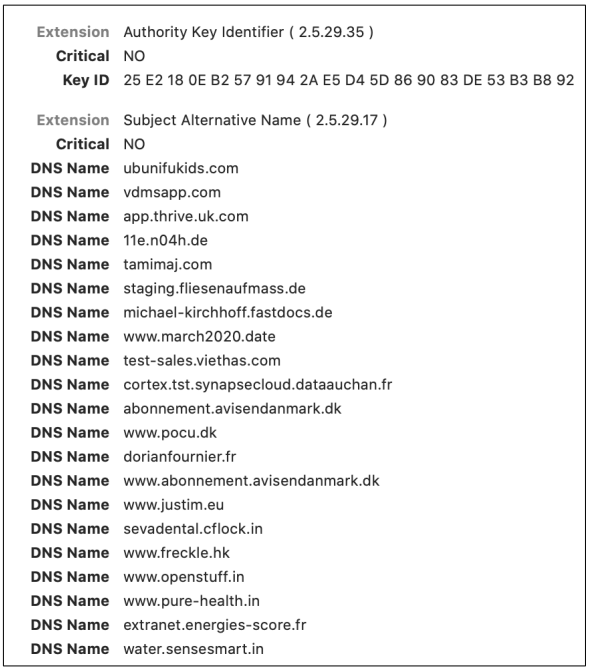
From these 159 million keys, there were around 600,000 duplicates. By repeating this process on the set of duplicates, we found that more than 10.000 keys had more

than five duplicates. This means that at least the same RSA keys were uploaded to the CT logs more than five times. However, it is more likely that the exact same certificate had been uploaded several times.

Investigating these duplicates led to finding certificates using X.509 version 2 and having the same certificate for many different and unrelated domains. Some of these certificates had over 45 different domains with no apparent connection. We think it may be linked to multi-domain certificates, but have no explanation for why X.509 version 2 was being used.

We visited some of the certificates associated domains to check their certificate currently in use in 2022. Here we found at least one certificate using X.509 version 3 that had 100 domains listed as a SAN extension. This can be seen in Figure 4.1. The purpose of a multi-domain certificate is to simplify the management of certificates for users who own multiple domains. As Digicert writes [12]:

> These certificates are ideal for securing many names across different domains and subdomains



Extension	Authority Key Identifier (2.5.29.35)
Critical	NO
Key ID	25 E2 18 0E B2 57 91 94 2A E5 D4 5D 86 90 83 DE 53 B3 B8 92
Extension	Subject Alternative Name (2.5.29.17)
Critical	NO
DNS Name	ubunifukids.com
DNS Name	vdmsapp.com
DNS Name	app.thrive.uk.com
DNS Name	11e.n04h.de
DNS Name	tamimaj.com
DNS Name	staging.fliesenaufmass.de
DNS Name	michael-kirchhoff.fastdocs.de
DNS Name	www.march2020.date
DNS Name	test-sales.viethas.com
DNS Name	cortex.tst.synapsecloud.dataauchan.fr
DNS Name	abonnement.avisendanmark.dk
DNS Name	www.pocu.dk
DNS Name	dorianfournier.fr
DNS Name	www.abonnement.avisendanmark.dk
DNS Name	www.justim.eu
DNS Name	sevadental.cflock.in
DNS Name	www.freckle.hk
DNS Name	www.openstuff.in
DNS Name	www.pure-health.in
DNS Name	extranet.energies-score.fr
DNS Name	water.sensesmart.in

Figure 4.1: Active X.509 Certificate issued by Google Trust Services found on a domain of an older certificate from our search which had many duplicate keys. This specific certificate had 100 "alternative subject name" with a wide variety of domains that seem completely unrelated

However, this also raises some questions that have yet to be answered. The questions we cannot seem to find answers to is, who controls the private key? Does every domain listed in the SAN field store a copy of the private key? Regardless of the answer to the latter question, how does the certificate serve as authentication of these domains? Another question is that of securing several domains, perhaps by a single RSA key. This should, of course, not be a problem as RSA is considered secure, but at least there are some risks to be considered. A naive comparison is to passwords, you can choose a password that will take hundred years to crack, but it is still considered a bad practice to reuse passwords. Certificates of this type could also be a more valuable target

for attacks as the "prize" is larger. It is also more impacting should such a certificate fail as multiple domains are affected. Again as RSA is considered secure, it should not be an issue but is still something to consider. Secondly, there could be some added risk to the user who may need to distribute the private key to all domains using the certificate. This could undoubtedly be achieved securely but what a user chooses to do or not is hard to control.

These concerns may be minor, but the most significant concern is that of ownership. Although not specified, the user should be one person or entity, as sharing a key with another entity or person could undermine security. At best, a key can be shared between two entities that trust each other and have good intentions, but as with any other password or other credentials, there is a risk involved when sharing them. At worst, the entities do not trust each other and may wish to harm each other, and thus have the ability to do so with the private key.

The certificate we found was issued by Google Trust Services and according to their subscriber agreement [17]:

> Subscriber will: (c) not use a Google-issued Certificate for or on behalf of any other person, organization, or entity;

And from their certificate policy [16]:

> The Certificate Warranties specifically include, but are not limited to, the following: 1. Right to Use Domain Name or IP Address: That, at the time of issuance, the CA (i) implemented a procedure for verifying that the Applicant either had the right to use, or had control of, the Domain Name(s) ... and subjectAltName extension (or, only in the case of Domain Names, was delegated such right or control by someone who had such right to use or control);

For this specific CA and their subscriber agreement, it is clear that the user cannot use the certificate on behalf of someone else. However, it is stated in the certificate policy, that using the certificate on behalf of someone who has the right to the domain is legal. This is the core of the issue of ownership.

The ownership issue is a matter between the CA and the user, but this question is very hard, if not impossible, to answer for a third party. Checking if some domains are owned or controlled by the same entity is no easy task. This also reduces the accuracy of the information on the certificate as the subject or user may not be the one the certificate is issued to. This is also a reason for not sharing a certificate between entities. The issue is the lack of transparency to third parties, thus removing the option of anyone finding vulnerabilities or bad practices. As this reduces the transparency and accuracy of the certificates, this seems to be a strange direction to move towards.

This short investigation left us with more questions than answers and can be a topic for further study.

Chapter 5

Conclusion

We wanted to repeat past experiments with collecting RSA keys and checking if any of them shared factors. So far the similar past experiments has mostly focused on collecting RSA keys by scanning the internet. The latest experiment was done in 2019 and was the first, to our knowledge, to collect keys from the CT logs, sampling 100 million keys. We only collected certificates from the CT logs and exclusively gathered 2048-bit keys. Our collection of 159,377,664 distinct RSA keys is, as far as we know, the largest such set used for checking of shared factors.

When running the algorithm the first time, we found eight keys with shared factors all issued by the same CA named ZeroSSL. This led us to collecting all the RSA keys issued by the ZeroSSL in the Argon2021 database and running the algorithm once more. This time we factored 355 unique keys, including the original eight, out of a set with more than 700,000 keys from ZeroSSL. After getting the results we contacted ZeroSSL which launched an investigation concluding that all keys were from one user who abused their browser-based generator by using an automated web browser. We cannot verify that all the domains belonged to one user, but we think it can make sense considering that most of the domains were Vietnamese or had Vietnamese sounding names. Considering that the user was maybe not a "common" user of their software we believe that ZeroSSL still has a vulnerability, as it is not clear why using an automated browser should result in weak keys.

Going forward, the continued analysis of the practice of cryptography through the CT logs still provides results. This thesis and the study from 2019 [25] has analysed less than 5% of the overall 6 billion certificates uploaded to the CT logs since 2013. Although there is a substantial difference in the results from the CT logs and from internet-wide scans.

One important lesson we learned doing this is the importance of storing the data systematically, as we did not use a database to store the certificates. We only stored the RSA modulus and the respective index from the Argon2021 database. Since we only operated on Argon 2021, this is a unique index which allowed us to easily fetch a particular certificate if needed. This made the process more troublesome than if we could simply sort and query specific requests.

Working with the CT logs has been fun, but there is definitely room for improvement on the servers. While understanding that it is an issue of bandwidth and load balancing, even the most basic task of retrieving entries takes a lot of time. Any sort of search or filter options would really help to investigate the CT logs, perhaps such features

would also decrease the overall requests. We sent some "unnecessary" requests while looking for specific certificates based on the algorithm used, issuer, key size etc. We also understand that this might not be their goal but either way it would help to be able to have some basic filter or search options to make the CT logs more accessible for research purposes.

If possible, the improvement to the batch GCD algorithm will allow larger and more thorough investigations of the state of cryptography in practice. The algorithm is very flexible in terms of resources, one could use a small computer with large amount of disk memory and get a large runtime but still be able to process large inputs. Alternatively, one needs less disk memory if there is more RAM available, this makes the algorithm would run faster. Finally, after running such an experiment one could, given more resources, keep adding new batches which would require less runtime to check. This way one could eventually, over time, check the entire CT logs and add new batches as the logs are updated.

The combination of improving or finding a better algorithm, and using more resources will make it possible to check a substantial amount of the active certificates from the CT logs. This may in turn discover vulnerabilities and make the overall web PKI more secure. As mentioned in Chapter 4, investigating and auditing the theory and practice of multi domain certificates may also improve the overall security and transparency of the web PKI, as we mostly found questions instead of answers when learning about them.

In the end, RSA is being replaced in the years to come by other public-key algorithms mainly due to RSA not being quantum safe. This means RSA can be broken by quantum computers, and as they are developed in the future, RSA will therefore not be secure. In TLS version 1.3 static RSA is removed as a key exchange algorithm, but will continue to be used for certificates and signatures [31]. So as people start using TLS 1.3 RSA will be less used. This shift could still take years, as people often use older versions until no longer supported or critical vulnerabilities are found. So RSA and the batch GCD algorithm will still be relevant for several years to come, and these kinds of experiments will still provide valuable insight, until the world has phased out RSA completely and transitioned to only using post quantum cryptography.

Bibliography

[1] Certificate transparency logs. https://certificate.transparency.dev, accessed 29.04.22.

[2] OpenMP. https://www.openmp.org/, accessed 12.05.22.

[3] C. Adams and S. Lloyd. *Understanding PKI*. Pearson, 2003.

[4] J. M. Alfred, C. V. O. Paul, A. V. Scott, and R. L. Rivest. *Handbook of applied cryptography*. CRC Press, 1997.

[5] H. Baker. How many atoms in the universe. https://www.livescience.com/how-many-atoms-in-universe.html, accessed 29.04.22.

[6] D. J. Bernstein. How to find the smooth parts of integers. https://cr.yp.to/factorization/smoothparts-20040510.pdf, accessed 03.05.22.

[7] CA/browser-forum. On extended validation. https://cabforum.org/extended-validation/, accessed 29.04.22.

[8] Clouflare. Certificate transparency log: Argon2021. https://ct.cloudflare.com/logs/argon2021, accessed 29.04.22.

[9] D. Cooper. rfc5280 X.509 certificates. https://datatracker.ietf.org/doc/html/rfc5280, accessed 03.05.22.

[10] J. Daemen and V. Rijmen. *The Design of Rijndael: AES The Advanced Encryption Standard*. Springer-Verlag, 2001.

[11] T. Dierks and E. Rescorla. rfc5246 on TLS 1.2. https://datatracker.ietf.org/doc/html/rfc5246#section-7.3, accessed 09.05.22.

[12] Digicert. Multi-Domain certificate. https://www.digicert.com/tls-ssl/multi-domain-ssl-certificates, accessed 29.04.22.

[13] P. ECKERSLEY and J. BURNS. An observatory for the SSLiverse. Talk at Defcon 18 (2010). https://www.eff.org/files/DefconSSLiverse.pdf, accessed 03.05.22.

[14] Google. Certificate Transparency in Chrome. https://github.com/GoogleChrome/CertificateTransparency, accessed 03.05.22.

[15] Google. Google ads. https://ads.google.com/intl/en/home/campaigns/app-ads/, accessed 29.04.22.

[16] Google. Google trust service's certificate policy. https://static.googleusercontent.com/media/pki.goog/no//repo/cps/4.8/GTS-CPS.pdf, accessed 02.05.22.

[17] Google. Google trust service's subscriber agreement. https://static.googleusercontent.com/media/pki.goog/no//repo/sa/1.0/GTS-SA.pdf, accessed 02.05.22.

[18] Google. Transparency report. https://transparencyreport.google.com/https/overview, accessed 29.04.22.

[19] T. Granlund. The GNU multiple precision arithmetic library. http://gmplib.org/, accessed 03.05.22.

[20] M. Hastings, J. Fried, and N. Heninger. Weak Keys Remain Widespread in Network Devices. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, pages 49–63, New York, NY, USA, 2016. Association for Computing Machinery.

[21] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[22] H. Hoogstraten. Black Tulip. https://roselabs.nl/files/audit_reports/Fox-IT_-_DigiNotar.pdf.

[23] ITU. About itu. https://www.itu.int/en/about/Pages/default.aspx, accessed 12.05.22.

[24] ITU. on x.500. https://www.itu.int/rec/T-REC-X.500, accessed 12.05.22.

[25] J. Kilgallin and R. Vasko. Factoring RSA Keys in the IoT Era. In *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 184–189, 2019.

[26] J. F. Kurose and K. W. Ross. *Networking: A top-down approach*. Pearson, 2017.

[27] S. Langkemper. RSA attacks. https://www.sjoerdlangkemper.nl/2019/06/19/attacking-rsa, accessed 29.04.22.

[28] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Public Keys. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 626–642, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[29] K. Moriarty. rfc8017 on RSA. https://datatracker.ietf.org/doc/html/rfc8017, accessed 29.04.22.

[30] T. Pornin. RSA: low public exponent. https://security.stackexchange.com/questions/2335, accessed 29.04.22.

[31] E. Rescorla. rfc8446 on TLS 1.3. https://datatracker.ietf.org/doc/html/rfc8446, accessed 29.04.22.

[32] R. Sears. Parsing certificate transparency lists. https://medium.com/cali-dog-security/parsing-certificate-transparency-lists-like-a-boss-981716dc506, accessed 24.04.22.

[33] D. R. Stinson. *Cryptography: theory and practice*. Chapman & Hall/CRC, 2006.

[34] P. Svenda, M. Nemec, P. Sekan, R. Kvasnovsky, D. Formanek, D. Komarek, and V. Matyas. The Million-Key Question Investigating the Origins of RSA Public Keys. In *The 25th USENIX Security Symposium (UsenixSec'2016)*, pages 893–910. USENIX, 2016.

[35] ZeroSSL. ZeroSSL's website. https://zerossl.com/about, accessed 29.04.22.

[36] P. Zimmermann. The Official PGP User's Guide, 1995.

[37] P. Zimmermann, F. Boudot, P. Gaudry, A. Guillevic, N. Heninger, and E. Thome. Factorization of RSA-250. https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;dc42ccd1.2002, accessed 03.05.22.